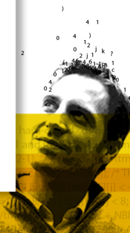
Frama-C Training Session

Browsing your code dependencies

Julien Signoles

CEA LIST

Software Reliability Labs



What are we going to do

How to better understand a C code within Frama-C
by extracting semantic information from this code

For what purpose

- helping to start verification of an unknown code
- helping to understand results of heavier analyses
- helping heavier analyses to give better results
- helping audit activities
- helping reverse-engineering activities

In what way

- using a battery of Frama-C plug-ins, either syntactic or semantic

Syntactic analyzers

Only deduce information from a direct use of the AST

Warnings

- those here-presented use the normalised program, not the original one
- does not use advanced semantical information (for instance, the value of a variable at some statement)
- in particular, does not handle pointers
- some may provide incorrect results in some cases

Syntactic analyzers within Frama-C

- analysing code using program syntax only is not the main goal of Frama-C
- only few syntactic analyzers in Frama-C

Syntactic analyzers

what they (do not) provide

What their are good for

- getting information quickly

What their are *not* good for

- providing a big amount of useful information
- providing confidence if they may provide incorrect results

Semantic analyzers

In this session, all semantic analyzers are based on abtract interpretation and the value analysis plug-in

Features

- theoretically sound: always provide correct results, as long as there are no soudness implementation bugs
- handle pointers correctly

Semantic analyzers within Frama-C

- most Frama-C plug-in are semantic analyzers

Warnings

- run the value analysis first
- may take a long time
- over-approximate the results
- all the ways to improve the efficiency/precision of the value analysis apply
- all the limitations of the value analysis also apply
- all the alarms emitted by the value analysis should be carefully examined

1. Lightweight analyzers
    - Metrics *syntactic*
    - Callgraphs *both*
    - Constant foldings *both*
    - Occurrence *semantic*
2. Dependencies and effects
    - Functional dependencies and effects *semantic*
    - Imperative effects *semantic*
    - Operational effect *semantic*
    - Data scoping *semantic*
3. Reducing code to analyse
    - Slicing *semantic*
    - Sparecode *semantic*
    - Impact *semantic*

They are either:

- syntactic analyzers; or
- semantic analyzers remaining quite precise even if the value analysis does not give so precise results

Metrics

syntactic code metrics

Give some syntactic metrics about the analyzed code.

Features

- defined and undefined functions
- number of calls to each function
- potential entry points (the never-called functions)
- number of loc
- number of conditionals, assignments, loops, calls, gotos, pointer access

Warnings

- measures are done on the normalised code, not on the original one
- does not take function pointers into account

Metrics
(continuing)

What is it good for

- helping to measure how difficult the analyses will be
- helping to identify whether some file is missing
- helping to identify which functions have to be stubbed or specified
- helping to identify entry points of the analyzed code

How to use

- -metrics dumps metrics on stdout
- -metrics-dump <f> dumps metrics on file f
- also (partially) available from the GUI

Syntactic callgraph

Indicate the callers of each function

Features

- representation as graphs into dot files
- notion of service, a group of related functions which seems to provide common functionalities

Warning

- does not take function pointers into account

What is it good for

- helping to identify entry points of the analyzed code
- helping to discover services provided by an application
- grasping the code architecture

How to use

- -cg <f> dumps callgraph in dot file f
- -cg-init-func <f> adds function f as a root service
- from the GUI: menu View, then Show Call Graph (still experimental)

Semantic callgraph

Same as the syntactic callgraph...
But using the program semantics

Features

- correctly deal with function pointers

Warnings

- run the value analysis first: may take a long time

What is it good for

- computing the callgraph for codes with function pointers

Semantic callgraph
(continuing)

How to use

- `-scg-dump` dumps the callgraph to stdout into dot format
- `-cg-init-func <f>` uses function `f` as a root service
- not available from the GUI

Warnings

- currently not the same interface as the syntactic callgraph (will be fixed soon)
- currently not exactly the same notion of service as the syntactic callgraph (will be fixed soon)

Users

function callees

Same as the semantic callgraph...
But not represented as a graph

Feature

- display the callees of each functions

Warning

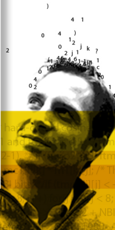
- no service computed

What is it good for

- extracting information with some external automatic tools (like grep)

How to use

- `-users` dumps the function callees on stdout



Syntactic constant folding

Fold all constant expressions in the code before analysis

Feature

- replace constant expressions by their results

Warning

- local propagation only: do not propagate the assignment of a constant to a left-value in the propram

What is it good for

- quickly simplifying programs with lots of constant expressions
- using analysis puzzled by big constant expressions

How to use

- `-constfold` performs this analysis before all others

Semantic constant folding

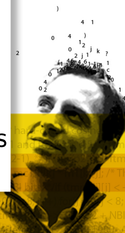
Propagate constant expressions in the whole program

More precisely

- generate a new program where expressions of the input program which are established as constant by the value analysis are
  - replaced by their value
  - propagated through the whole program

Features

- the output program is a compilable C code
- it has the same behaviour as the original one
- handle constant integers and pointers, even function pointers



Semantic constant folding
(continuing)

Warning

- does not handle floating-point values yet

What is it good for

- simplifying programs with lots of constant values
- using analysis puzzled by constant expressions

How to use

- `-semantic-const-folding` propagates constants and pretty print the new source code
- `-semantic-const-fold <f1>, ..., <fn>` propagates constants only into functions `f1`, ..., `fn`
- `-cast-from-constant` replaces expressions by constants even when doing so requires a pointer cast

Occurrence

where variables are used

Show the uses of a variable in a program

More precisely

- highlight the left-values that may access a part of the location denoted by the selected variable

Features

- take aliasing into account
- also show uses of a C variable in logic annotations
- mainly a graphical plug-in

Warnings

- quite difficult to use in batch mode
- does not handle logic variable yet

Occurrence
(continuing)

What is it good for

- understanding a quite mysterious piece of code
- discovering some unknown aliases of the program

How to use

- `-occurrence` dumps the occurrences of each variable on stdout
- from the GUI: left panel and contextual menu

Outline

1. Lightweight analyzers
   - Metrics *syntactic*
   - Callgraphs *both*
   - Constant foldings *both*
   - Occurrence *semantic*
2. Dependencies and effects
   - Functional dependencies and effects *semantic*
   - Imperative effects *semantic*
   - Operational effects *semantic*
   - Data scoping *semantic*
3. Reducing code to analyse
   - Slicing *semantic*
   - Sparecode *semantic*
   - Impact *semantic*

Dependencies and effects

Features

- several notions of input/output for functions
- several kinds of dependencies

Functional dependencies

Dependencies between inputs and outputs of functions

Definitions

- functional output of a function `f`: left-value that may be modified in `f` when `f` terminates
- functional input of a function `f`: left-value which may impact the output value of a functional output of `f`

Features

- functional outputs and inputs
- dependencies between outputs and inputs
- indicate whether the analyzer knows that an output is always modified (when the function terminates)
- ignore local variables (from the next release)

Functional dependencies
(continuing)

How to use

- mainly a batch plug-in
- `-deps` displays the functional dependencies for each function
- `-calldeps` displays the functional dependencies by callsite: if a function is called several times, results are not merged

What is it good for

- providing dataflow specifications of functions
- helping to understand relations between inputs and outputs of each function
- improving precision of other analyser through `-calldeps`

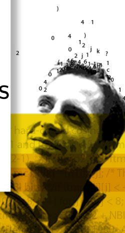
Inout

imperative and operational effects

What is read, what is written,
what is read before being written

Definitions

- imperative input of a function f: left-value that may be read in f
- imperative output of a function f: left-value that may be written in f
- operational input of a function f: left-value that is read without having been previously written to, when f terminates



Inout

(continuing)

Features

- imperative inputs and outputs
- operational inputs

Warnings

- mainly a batch plug-in
- operational inputs are still experimental: the specification may change
- operational outputs exist but are not yet documented

Inout

command line options

How to use

- -input displays the imperative inputs of each function; locals and function parameters are not displayed
- -input_with_formals same as -input, but displaying function parameters
- -out displays the imperative outputs of each function
- -inout displays the operational inputs of each function

Scope

data scoping

Dependencies of a given left-value l at a given program point L

Features

- show defs: statements that may define the value of l at L
- zones: statements that may contribute to define the value of l at L
- data scope: statements where l is guaranteed to have the same value as at L

Warning

- still experimental

Scope
(continuing)

What is it good for

- locally better understand what the program does
  - relations between left-values
  - where the current value of a left-value comes from
  - scope of definition of a left-value

How to use

- only available from the GUI: sub-menu Dependencies of the contextual menu with three entries (Show defs, Zones, DataScope)

Outline

1. Lightweight analyzers
   - Metrics *syntactic*
   - Callgraphs *both*
   - Constant foldings *both*
   - Occurrence *semantic*

2. Dependencies and effects
   - Functional dependencies and effects *semantic*
   - Imperative effects *semantic*
   - Operational effects *semantic*
   - Data scoping *semantic*

3. Reducing code to analyse
   - Slicing *semantic*
   - Sparecode *semantic*
   - Impact *semantic*

Reducing code to analysed

Features

- generate a new program in a new project
- the new program is compilable
- the new program is usually shorter
- the new program is usually easier to analyze

Warning

- usually is not always...

Specialize the program according to some user-provided criteria

Features

- generate a new program in a new project
- the new program is compilable
- the new program and the analysed one have the same behaviour according to the slicing criterion

What are the available criteria?

Criteria for code observation

- preserving effects of statements
- preserving the read/write accesses of/to left-values

Criteria for proving properties

- preserving behaviour of assertions
- preserving behaviour of loop invariants
- preserving behaviour of loop variants
- preserving behaviour of threats (emitted by the value analysis)

Pragmas

- `/*@ slice pragma ctrl; */` preserves the reachability of this control-flow point
- `/*@ slice pragma expr e; */` preserves the value of the ACSL expression `e` at this control-flow point
- `/*@ slice pragma stmt; */` preserves the effects of the next statement

How to use

- from command line options
- from the GUI: left panel and contextual menu

Slicing

command line options

Each option preserves the semantics of the input program according to a specific criterion

Options of the form -slice-*criterion* <f1>, ..., <fn>

- -slice-calls: calls to these functions
- -slice-return: the return of the these functions
- -slice-pragma: slicing pragmas in theses functions
- -slice-assert: assertions of these functions
- -slice-loop-inv: loop invariants in these functions
- -slice-loop-var: loop variants in these functions
- -slice-threat: threats in these functions

Software Analyzers

Slicing

command line options (continuing)

Options of the form -slice-*criterion* <v1>, ..., <vn>

- -slice-value values of these left-values at the end of the entry point
- -slice-rd read access to these left-values
- -slice-wr write access to these left-values

Warning

- addresses of the left-values are evaluated at the beginning of the entry point

Custom options

- -slicing-level <n> specifies how to slice the callees
  - 0: never slice the called functions
  - 1: slice the callees but preserves all their functional outputs
  - 2: slice the callees but create at most 1 slice by function
  - 3: most precise slices; create as many slices as necessary

  Default level is 2
- -no-slice-undef-functions does not slice the prototype of undefined functions (default)
- -slice-undef-functions slices the prototype of undefined functions
- -slice-print pretty prints the sliced code

Warning

- the higher the slicing level is, the slower the slicing is

What is it good for

- helping to extract the signifiant parts of a program according to your own criteria
- helping to understand where a behavior comes from
- helping analyses to give better results
- helping audit activities

Sparecode

code cleaner

Remove useless code of the program

Features

- generate a new program in a new project
- the new program is compilable
- the values assigned to the output variables of the main function are preserved in the new program
- slicing pragmas may be used to keep some statements
  - `/*@ slice pragma ctrl; */`
  - `/*@ slice pragma expr e; */`
  - `/*@ slice pragma stmt; */`

Sparecode
(continuing)

Warnings

- still experimental
- partial support of ACSL: only the annotations inside function bodies (*e.g.* assertions) are processed at the moment; all the others are ignored and do not appear in the new program

What is it good for

- help to discover what is useless in a program
- may improve the results of others analyzers which are puzzled by some useless code

How to use

- `-sparecode-analysis` removes statements and functions that are not useful to compute the result of the program
- `-rm-unused-globals` removes unused types and global variables
- `-sparecode-no-annot` may remove some useless code even if it changes the validity of some ACSL properties

Impact

What could be discovered
if the side effect of a statement would be revealed

More precisely

- a statement `s` is impacted by a statement `s'` iff modifying the effect of `s'` by another *possible* one may modify the effect of `s`
- an effect is *possible* iff there is an execution of the program that generates this effect. For instance, the possible effects of `z=x+y;` in `x=c?0:1; y=c?0:1; z=x+y` are `z` becomes equal to `0` or `2`.

Warning

- still experimental

Impact
(continuing)

What is it good for

- helping to understand what a statement is useful for
- helping to apprehend code changes
- helping audit activities, in particular security audits

How to use

- `-impact-pragma <f1>, ..., <fn>` computes the impact from the pragmas in functions `f1`, ..., `fn`. Only the following pragma is yet usable.
  `/*@ impact pragma stmt; */`
- `-impact-print` dumps the result of the analysis on stdout
- from the GUI: left panel and contextual menu

Battery of Frama-C plug-ins presented

For what purpose

- helping to start verification of an unknown code
- helping to understand results of heavier analyses
- helping heavier analyses to give better results
- helping audit activities
- helping reverse-engineering activities

Browse your code dependencies more easily!